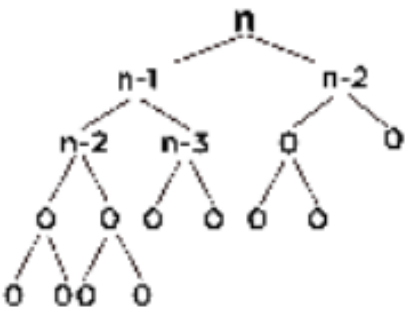
Dynamic Programming - I

Introduction

Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
public static int fibo(int n){
      if(n <= 1)
            return n;
      return fibo(n-1) + fibo(n-2);
}
```

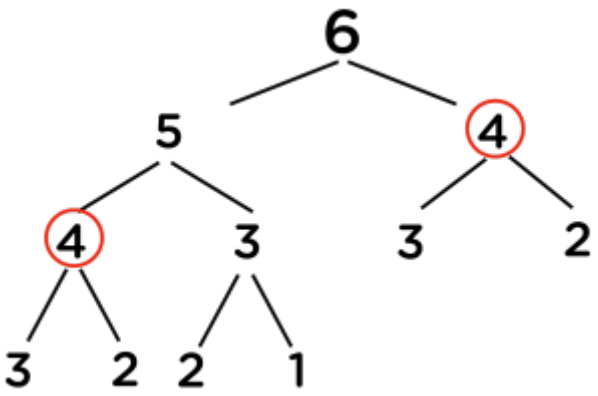
- Here, for every n, we need to make a recursive call to f(n-1) and f(n-2).
- For f(n-1), we will again make the recursive call to f(n-2) and f(n-3).
- Similarly, for f(n-2), recursive calls are made on f(n-3) and f(n-4) until we reach the base case.
- The recursive call diagram will look something like shown below:



- At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one).
- At every level, we are doing (2^n)*K work (where n = 0, 1, 2, ...).
- Since reaching 1 from n will take n calls, therefore, at the last level, we are doing (2^(n-1))*k work.
- Total work can be calculated as

$$(2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}) * k \quad = \quad 2^n k$$

- Hence, it means time complexity will be O(2n).
- We need to improve this complexity. Let's look at the example below for finding the 6th Fibonacci number.



Important Observation:

- We can observe that there are repeating recursive calls made over the entire program.
- As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4)(second similar recursive call over f(4)).
- Both of these recursive calls are shown above in the outlining circle.
- Similarly, there are many other values for which we are repeating the recursive calls.
- Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values(preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again. This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called Memoization.

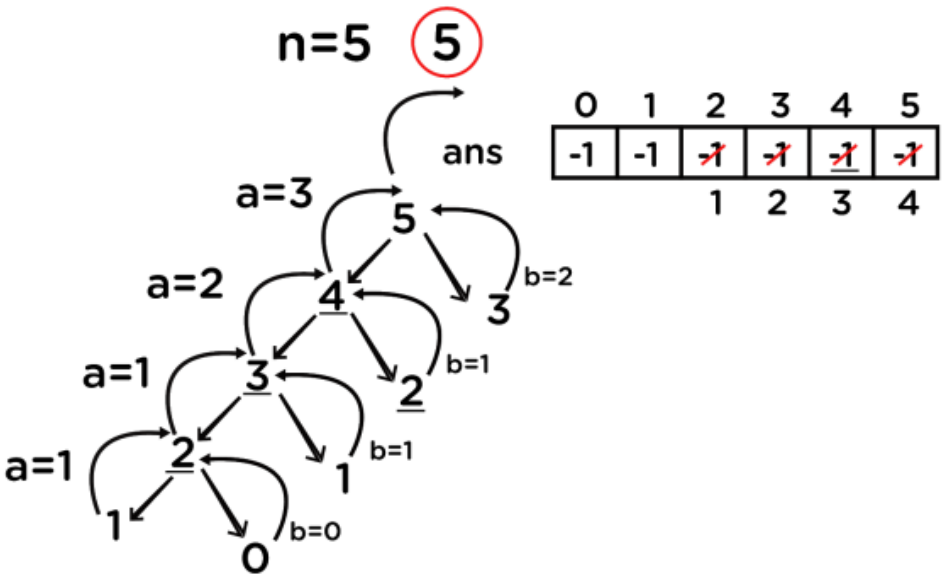
- To achieve this in our example we will simply take an answer array, initialised to -1.
- Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not.
- If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value.

- After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most (n+1) only. Let's look at the memoization code for Fibonacci numbers below:

```
private static int fibo_helper(int n, int[] ans){
      if (n==0 || n==1)       //Base case
            return n;

      //check if output already exists
      if (ans[n] != -1){
            return ans[n];
      }

      // calculate output
      int a = fibo_helper(n-1, ans);
      int b = fibo_helper(n-2, ans);

      // save the output for future use
      ans[n] = a + b;

      // return the final output
      return ans[n];
}

public static int fibo_2(int n){
      int[] ans = new int[n+1];

     for(int i=0; i<=n; i++){
           ans[i] = -1;       // -1 represents that fibb for that
     }                        // index does not exist

      return fibo_helper(n, ans);
}
```

Let's dry run for n = 5, to get a better understanding



Again, if we observe carefully, we can see that for any number, we are not able to make a recursive call on the right side of it. This means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimised as compared to simple recursion.

Summary

- Memoization is a top-down approach, where we save the previous answers so that they can be used to calculate future answers and improve the time complexity to a greater extent.
- Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored.
- Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes.
- In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index.
- This can be simply done iteratively by running a loop from i = (2 to n).
- Finally, we will get our answer at the 5th index of the answer array as we already know that the i-th index contains the answer to the

i-th value. We are first trying to figure out the dependency of the current value on the previous values and then using them to calculate our new value.

Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, we will follow a bottom-up approach to reach the desired index. This approach of converting recursion into iteration is known as Dynamic programming(DP).Let us now look at the DP code for calculating the nth Fibonacci number:

```
public static int fibo_3(int n){
     int[] ans = new int[n+1];

     ans[0] = 0;     // storing independent values in solution array
     ans[1] = 1;

     // following bottom-up approach to reach n
     for(int i=2; i<=n; i++){
          ans[i] = ans[i-1] + ans[i-2];
     }

     return ans[n];     // final answer
}
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach. For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimise the recursive approach by storing the previous answers using memoization.
3. Finally, replace recursion by iteration using dynamic programming. (It is preferred to be done in this manner because recursion generally has an increased space complexity as compared to iteration methods.)

Problem Statement: Frog Jump

Explanation : There is a frog on the '1st' step of an 'N' staircase. The frog wants to reach the 'Nth' stair. 'HEIGHT[i]' is the height of the '(i+1)th' stair.If Frog jumps from 'ith' to 'jth' stair, the energy lost in the jump is given by absolute value of (HEIGHT[i-1] - HEIGHT[j-1]). If the Frog is on 'ith' staircase, he can jump either to '(i+1)th' stair or to '(i+2)th' stair. Your task is to find the minimum total energy used by the frog to reach from '1st' stair to 'Nth' stair.

Example : If the given ‘HEIGHT’ array is [10,20,30,10], the answer 20 as the frog can jump from 1st stair to 2nd stair (|20-10| = 10 energy lost) and then a jump from 2nd stair to last stair (|10-20| = 10 energy lost). So, the total energy lost is 20.

Approach - I

In this approach, we will define a recursive function REC(i,HEIGHT) that will return the minimum energy needed to reach the last stair from the ith stair.The base case will be if i is greater than or equal to ‘N’ answer will be 0 as we already reached the final stair.As we have two choices at each step,REC(i) will be the maximum of energy lost for jumping from ith to (i+1)th step + REC(i+1) and energy lost for jumping from i th to (i+2)th step + REC(i+2).

The final answer will be REC(1, HEIGHTS) corresponding to the minimum energy required to reach the last stair from the first stair.

Algorithm:

- Defining 'REC'(i,' HEIGHTS’) function :
    - If i is equal to the length of ‘HEIGHTS’ - 1:
        - Return 0.
    - Set ‘ONE_JUMP’ as INF.
    - Set ‘TWO_JUMP’ as INF.
    - If i+1 < length of ‘HEIGHTS’:
        - Set ‘ONE_JUMP’ as abs(HEIGHTS[i]-HEIGHTS[i+1]) + REC(i+1,HEIGHTS).
    - If i+2 < length of ‘HEIGHTS’:
        - Set ‘TWO_JUMP’ as abs(HEIGHTS[i]-HEIGHTS[i+2]) + REC(i+2,HEIGHTS).
    - Set ‘ANS’ as minimum of ONE_JUMP and TWO_JUMP.
    - Return ‘ANS’.
- Set ‘ANS’ as REC(1,HEIGHTS).
- Return ‘ANS’.

Optimised Approach :

In this approach, we will make an array DP of size N+1.DP[i] will denote the minimum energy required to reach the last stair from the ith stair.The base case will be DP[N] should be equal to zero, as the frog is already at the last stair. DP[N-1] will be abs(HEIGHTS[N-1] - HEIGHTS[N-2]) as only one jump is possible. We will run a loop from N-2 to 1 to compute the values of DP[i] as follows:

Choice 1: will be jumping from i to i+1,So DP[i] will be DP[i+1] + abs(HEIGHTS[i-1]- HEIGHTS[i]).

Choice 2 : will be jumping from i to i+2,So DP[i] will be DP[i+2] + abs(HEIGHTS[i-1]- HEIGHTS[i+1]).

For all i, we will pick the minimum of these two choices.

At last, we will return DP[1] as the final answer.

Algorithm:

- Declare an array 'DP' of size N+1.
- Set DP[N] as 0.
- Set DP[N-1] as abs(HEIGHTS[N-1] - HEIGHTS[N-2]).
- For i in range N-2 to 1:
    - Set ONE_JUMP as DP[i+1] + abs(HEIGHTS[i-1]-HEIGHTS[i]).
    - Set TWO_JUMP as DP[i+2] + abs(HEIGHTS[i-1]-HEIGHTS[i+1]).
    - Set DP[i] as the minimum of ONE_JUMP and TWO_JUMP.
- Set 'ANS' as DP[1].
- Return ANS.

Problem Statement : Unique Paths

Explanation : Given a 'N' * 'M' maze with obstacles, count and return the number of unique paths to reach the right-bottom cell from the top-left cell. A cell in the given maze has a value '-1' if it is a blockage or dead-end, else 0. From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only. Since the answer can be large, print it modulo 10^9 + 7.

Example :

```
Consider the maze below :
0 0 0
0 -1 0
0 0 0

There are two ways to reach the bottom left
corner -

(1, 1) -> (1, 2) -> (1, 3) -> (2, 3) -> (3,
3)
(1, 1) -> (2, 1) -> (3, 1) -> (3, 2) -> (3,
3)

Hence the answer for the above test case is
2.
```

Approach : We can observe that we can reach cell (i, j) only through two cells if they exist -(i - 1, j) and (i, j - 1). Also if the cell (i, j) is blocked we can simply calculate the number of ways to reach it as zero.So if we know the number of ways to reach cell (i - 1, j) and (i, j - 1) we can simply add these two to get the number of ways to reach cell (i, j). Let paths(i, j) be the number of ways to reach cell (i, j).So our recursive relation can be defined as follows:

paths(i, j) = 0 if the cell is blocked or invalid.

paths(i, j) = paths(i - 1, j) + (i, j -1) otherwise.

Algorithm:

1. Create a recursive function paths(i, j) that will return the number of ways to reach the cell (i, j).
2. If the cell is invalid or blocked, return 0.
3. If the cell is (0, 0) return 1.
4. Else return (paths(i - 1, j) + paths(i, j - 1))%mod recursively.

Approach:

We can observe that we are calculating the same value, again and again, using recursion. For example paths(2, 3) and paths(3, 2) will both use paths(2, 2) which are being calculated separately twice. We can save this time by using dynamic programming and saving the results for previous paths.

Algorithm:

1. Create a 2D matrix dp of the same size as the given matrix to store the value of paths(i, j).
2. Traverse through the created array row-wise and start filling the values in it.
3. If an obstacle is found, set the value to 0.
4. For the first row and column, set the value to 1 if an obstacle is not found.
5. Set the sum of the right and the upper values if an obstacle is not present at that corresponding position in the given matrix
6. Return the value of dp[n][m] finally.

Time Complexity: O(N*M) Where N and M are dimensions of the input matrix.

We calculate the value of dp[i][j] of a cell in O(1) time, and since there are O(N*M) cells we calculate the table in O(N*M) time. Hence the time complexity is O(N*M).

Space Complexity: O(N*M) Where N and M are dimensions of the input matrix.

We create a dp table of size N*M. Hence the space complexity is O(N*M).

Problem Statement: TRIANGLE

Explanation : You are given a triangular array/list 'TRIANGLE'. Your task is to return the minimum path sum to reach from the top to the bottom row.The triangle array will have N rows and the i-th row, where 0 <= i < N will have i + 1 elements.You can move only to the adjacent number of rows below each step. For example, if you are at index j in row i, then you can move to i or i + 1 index in row j + 1 in each step.

For Example :

If the array given is 'TRIANGLE' = [[1], [2,3], [3,6,7], [8,9,6,1]] the triangle array will look like:

1
2,3
3,6,7
8,9,6,10

For the given triangle array the minimum sum path would be 1->2->3->8. Hence the answer would be 14. We can make a two dimensional 'DP' table and we know the base cases as now we are filling our 'DP' table in a bottom-up manner. Below are the steps :

- Initialise a two-dimensional array of size 'N'*'N' .
- Start with the bottom-most row of triangle and fill your 'DP' table , for the last row 'DP[N-1][POS]' = 'ARR[N-1][POS]' ; for all 'POS' from 0 to 'N'-1;
- Start filling your 'DP' table this way , 'DP[i][POS]' = min('DP[i+1][POS]' , 'DP[i+1] [POS+1]') + 'ARR[i][POS]', for all 'POS' from 0 to 'ROWSIZE' - 1
- Fill it for all rows till 0th row using the recurrence

Note: 'DP[0][0]' gives the result in bottom-up fashion.

Time Complexity: O(N * N), where N is the size of the triangle array.

As every element of triangle array will be visited at most once and there are N^2 elements in total, hence the overall time complexity will be O(N * N).

Space Complexity: O(N * N), where N is the size of the triangle array.

The space complexity due to the triangle array will be O(N * N).

Problem statement: Maximum Path Sum in the matrix

Explanation: You have been given an N*M matrix filled with integer numbers, find the maximum sum that can be obtained from a path starting from any cell in the first row to any cell in the last row.
From a cell in a row, you can move to another cell directly below that row, or diagonally below left or right. So from a particular cell (row, col), we can move in three directions i.e. Down: (row+1,col),Down left diagonal: (row+1,col-1),Down right diagonal: (row+1, col+1)

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

The maximum path sum will be 2->100->1->2, So the sum is 105(2+100+1+2).
In the second test case for the given matrix, the maximum path sum will be 10->7->8, So the sum is 25(10+7+8).

Approach : We are given an N*M matrix. Our idea behind this approach is while traversing in the given matrix, and we will keep updating the matrix with the best sum path till now. That means, we are going to the next row after completing the current row, So for each cell, all the cells from where we can reach here already contain the best path sum from any cell of the first row to that cell. Let's say we are at (row, col), So there is only three way to reach (row, call):

1. From up:(row-1, col)

2. From up left diagonal:(row-1,col-1)
3. From upright diagonal:(row-1,col+1)

So, matrix[row][col] will be,

matrix[row][col] = matrix[row][col] + max(matrix[row - 1][col], max(matrix[row - 1][max(0, col - 1)], matrix[row - 1][min(m-1, col + 1)]));

Where 'M' is the number of columns in the given matrix.

Once we have updated the "matrix" then each cell of the matrix (row, col) will represent the maximum path sum for any cell from the first row to the current (row, col) cell. So we will traverse in the last row and find the maximum value that will be the maximum path sum from any cell of the first row to any cell of the last row.

Time Complexity: O(N*M), Where 'N' is the number of rows and 'M' is the number of columns in the given matrix.
There will be a total N*M cell in the given matrix, and we are traversing each cell. So time complexity will be O(N*M).

Space Complexity: O(1) Since we are not using any extra space, we are just modifying the given matrix. So space complexity will be O(1).